4 Datenverwaltung

lernen

Programme tun ja eigentlich nichts anderes, als Daten zu verwalten und damit zu arbeiten. Auf der einen Seite haben wir die bereits besprochenen Methoden, in denen wir Anweisungen zusammenfassen können, die etwas mit unseren Daten tun. Auf der anderen Seite stehen die Daten selbst. In diesem Kapitel wollen wir uns nun mit den grundlegenden Datentypen des .NET Frameworks beschäftigen und aufzeigen, wie man damit arbeitet.

4.1 Datentypen

4.1.1 Speicherverwaltung

Arten von Datentypen

C# kennt zwei Sorten von Datentypen, nämlich einmal die wertebehafteten Typen, kurz auch *Wertetypen* genannt, und dann die *Referenztypen*. Der Unterschied besteht in der Art, wie die Werte gespeichert werden. Während bei Wertetypen der eigentliche Wert direkt gespeichert wird, speichert ein Referenztyp lediglich einen Verweis auf das betreffende Objekt. Dieses liegt irgendwo im Speicher, die Variable weiß wo und kann die Daten liefern. Wertetypen werden in C# grundsätzlich auf dem so genannten *Stack* gespeichert, Referenztypen auf dem so genannten *Heap*.

Stack

Als Programmierer müssen Sie sich des Öfteren mit solchen Ausdrücken wie *Stack* und *Heap* herumschlagen, aus diesem Grund hier auch die Erklärung, auch wenn sie eigentlich erst bei wirklich komplexen Programmierproblemen eine Rolle spielen. Als Stack bezeichnet man einen Speicherbereich, in dem Daten einfach abgelegt werden, solange sie gebraucht werden, z. B. bei lokalen Variablen oder Methodenparametern. Jedes Mal, wenn eine neue Methode aufgerufen oder eine Variable deklariert wird, wird eine Kopie der Daten erzeugt.

Freigegeben werden die Daten in dem Moment, in dem sie nicht mehr benötigt werden. Das bedeutet, in dem Moment, in dem Sie eine Variable deklarieren, wird Speicher reserviert in der Größe, die dem maximalen Wert entspricht, den die Variable enthalten kann. Dieser ist festgelegt über die Art des Datentyps, z. B. 32 Bit (4 Byte) beim Datentyp `int`.

Heap

Mit dem Heap sieht es ganz anders aus. Speicher auf dem Heap muss angefordert werden und kann, wenn er nicht mehr benötigt wird, auch wieder freigegeben werden. Das darin enthaltene Objekt wird dabei gelöscht. Wenn Sie Instanzen von Klassen erzeugen, wird der dafür benötigte Speicher beim Betriebssystem angefordert und dieses kümmert sich darum, dass Ihr Programm den Speicher auch bekommt.

Programmiersprachen wie z. B. C++ erfordern, dass der angeforderte Speicher explizit wieder freigegeben wird, d. h es wird darauf gewartet, dass Sie selbst im Programm die entsprechende Anweisung dazu geben. Geschieht dies nicht, kommt es zu so genannten *Speicherleichen*, d. h. Speicher ist und bleibt reserviert, obwohl das Programm, das ihn angefordert hat, längst nicht mehr läuft. Ein weiteres Manko ist, dass bei einem Neustart des Programms vorher angeforderter Speicher nicht mehr erkannt wird – wenn Sie also vergessen, Speicher freizugeben, wird irgendwann Ihr Betriebssystem die Meldung »Speicher voll« anzeigen und eine weitere Zusammenarbeit verweigern.

4.1.2 Die Null-Referenz

Der Wert `null` ist der Standardwert für alle Referenztypen. Wenn diese zwar deklariert, aber noch nicht instanziert sind, haben sie den Wert `null`. Eigentlich handelt es sich dabei um eine Referenz ins »Leere«, die Sie auch kontrollieren können.

null

Nehmen wir an, Sie hätten eine Methode geschrieben, die ein Objekt zurückliefern soll. Das ist ja ohne weiteres möglich, denn der Ergebniswert einer Methode kann frei festgelegt werden. Es soll aber im Fehlerfall, wenn die Methode nicht korrekt durchlaufen wird, ein Wert zurückgeliefert werden, der der aufrufenden Methode signalisiert, dass das verlangte Objekt nicht erzeugt werden konnte.

Die einfachste Möglichkeit ist, in diesem Fall einfach `null` zurückzuliefern. In der aufrufenden Methode kann der Wert dann kontrolliert und, falls er `null` ist, eine entsprechende Aktion eingeleitet werden.

Sie haben auch die Möglichkeit, ein bestehendes Objekt mithilfe von null zu dereferenzieren, d.h. klarzumachen, dass dieses Objekt jetzt nicht mehr existiert. Wenn Sie einem Objekt null zuweisen, besitzt es keine Referenz mehr, es kann nicht darauf zugegriffen werden.

Wenn Sie einem Objekt den Wert `null` zuweisen, wird zwar die Referenz darauf entfernt, das Objekt selbst existiert aber noch. Damit ist gemeint, dass der Speicher, der von betreffendem Objekt belegt wurde, noch nicht freigegeben wurde. Das geschieht erst beim Durchlauf der Garbage-Collection. Die erkennt, dass auf das Objekt keine Referenz mehr existiert und es dann aus dem Speicher entfernt.

4.1.3 Garbage-Collection

Mit C# hat die Angst vor Speicherleichen ein Ende, was auch bereits in der Einführung angesprochen wurde. Das .NET Framework, also die Basis für C# als Programmiersprache, bietet eine automatische *Garbage-Collection*, die nicht benötigten Speicher auf dem Heap automatisch freigibt. Der Name bedeutet ungefähr so viel wie »Müllabfuhr«, und genau das ist auch die Funktionsweise – der »Speichermüll« wird abtransportiert.

new

In C# werden Sie deshalb kaum einen Unterschied zwischen Wertetypen und Referenztypen feststellen, außer dem, dass Referenztypen stets mit dem reservierten Wort `new` erzeugt werden müssen, während bei Wertetypen in der Regel eine einfache Zuweisung genügt.

object

Hinzu kommt, dass alle Datentypen in C# wirklich von einer einzigen Klasse abstammen, nämlich der Klasse `object`. Das bedeutet, dass Sie nicht nur Methoden in anderen Klassen implementiert haben können, die mit den Daten arbeiten, vielmehr besitzen die verschiedenen Datentypen selbst bereits einige Methoden, die grundlegende Funktionalität bereitstellen. Dabei ist es vollkommen egal, ob es sich um Werte- oder Referenztypen handelt.

4.1.4 Methoden von Datentypen

Wie wir in Kapitel 3 bereits gesehen haben, gibt es zwei verschiedene Arten von Methoden, nämlich einmal die *Instanzmethoden*, die nur für die jeweilige Instanz des Datentyps gelten, und dann die *Klassenmethoden* oder *statischen Methoden*, die Bestandteil der Klasse selbst sind und sich nicht um die erzeugte Instanz scheren.

Parse()

So ist die Methode `Parse()` z. B. eine Klassenmethode. Wenn Sie nun eine Variable des Datentyps `int` deklariert haben, können Sie die Methode `Parse()` dazu verwenden, den Inhalt eines Strings in den Datentyp `int` umzuwandeln. Diese Methode ist in allen numerischen Datentypen implementiert, falls Sie also einen 32-Bit-Integer-Wert verwenden wollen, sähe die Deklaration folgendermaßen aus:

```
int i = Int32.Parse(myString);
```

Alternativ könnten Sie auch die entsprechende statische Methode der Klasse `Convert` benutzen. Diese Klasse enthält eine große Anzahl Methoden zur Konvertierung von Datentypen. Wenn man `Convert` benutzt, sieht der Aufruf dann so aus:

```
int i = Convert.ToInt32(myString);
```

Der Unterschied zwischen beiden Methoden ist auf den ersten Blick nicht ersichtlich, tun doch beide im Prinzip das Gleiche. `Parse()` allerdings berücksichtigt auch die landesspezifischen Einstellungen des Betriebssystems. Daher ist sie im Allgemeinen vorzuziehen. Mehr zu den Umwandlungsmethoden und zu `Parse()` noch in *Kapitel 4.2.5*.

Typsicherheit

C# ist eine typsichere Sprache, und somit sind die Datentypen auch nicht frei untereinander austauschbar. In C++ konnte man z. B. die Datentypen `int` und `bool` sozusagen zusammen verwenden, denn jeder Integer-Wert größer als 0 lieferte den booleschen Wert `true` zurück. In C# ist dies nicht mehr möglich. Hier ist jeder Datentyp autonom, d. h. einem booleschen Wert kann kein ganzzahliger Wert zugewiesen werden. Stattdessen müssten Sie, wollten Sie das gleiche Resultat erzielen, eine Kontrolle durchführen, die dann einen booleschen Wert zurückliefert. Wir werden im weiteren Verlauf dieses Kapitels noch ein Beispiel dazu sehen.

Wert- und Typumwandlung

Dennoch kann ein Datentyp auf mehrere Arten in einen anderen Datentyp umgewandelt werden. Eine Möglichkeit, z. B. aus einem `String`, der eine Zahl enthält, einen Integer-Wert zu machen, haben wir bereits kennen gelernt. Allerdings ist es aufgrund der Typsicherheit auch so, dass sogar zwei numerische Datentypen nicht einander zugeordnet werden können, wenn z. B. der Quelldatentyp einen größeren Wertebereich als der Zieldatentyp besitzt. Hier muss eine explizite Konvertierung stattfinden, ein so genanntes *Casting*, wodurch C# gezwungen wird, die Datentypen zu konvertieren. Dabei handelt es sich also um eine Wertumwandlung, während das obige Beispiel eine Typumwandlung darstellt.

Anders herum – von einem Datentyp mit einem kleinen Wertebereich in einen Datentyp mit einem größeren Wertebereich – funktioniert es anstandslos. Dennoch wird auch hier eine Konvertierung durchgeführt, eine so genannte implizite Konvertierung, die der Compiler automatisch durchführt.

Damit genug zur Einführung. Kümmern wir uns nun um die Standard-Datentypen von C#.

4.1.5 Standard-Datentypen

Einige Datentypen haben wir schon kennen gelernt, darunter der Datentyp `int` für die ganzen Zahlen und der Datentyp `double` für die reellen Zahlen. Alle diese Datentypen sind unter dem Namensraum `System` deklariert, den Sie in jedes Ihrer Programme mittels `using` einbinden sollten. Tabelle 4.1 gibt Ihnen nun einen Überblick über die Standard-Datentypen von C#.

Alias	Größe	Bereich	Datentyp in System
`sbyte`	8 Bit	-128 bis 127	`SByte`
`byte`	8 Bit	0 bis 255	`Byte`
`char`	16 Bit	Nimmt ein 16-Bit Unicode-Zeichen auf	`Char`
`short`	16 Bit	-32768 bis 32767	`Int16`
`ushort`	16 Bit	0 bis 65535	`UInt16`
`int`	32 Bit	-2147483648 bis 2147483647	`Int32`
`uint`	32 Bit	0 bis 4294967295	`UInt32`
`long`	64 Bit	–9223372036854775808 bis 9223372036854775807	`Int64`
`ulong`	64 Bit	0 bis 18446744073709551615	`UInt64`
`float`	32 Bit	$\pm 1.5 \times 10^{-45}$ bis $\pm 3.4 \times 10^{38}$ (auf 7 Stellen genau)	`Single`
`double`	64 Bit	$\pm 5.0 \times 10^{-324}$ bis $\pm 1.7 \times 10^{308}$ (auf 15-16 Stellen genau)	`Double`
`decimal`	128 Bit	1.0×10^{-28} bis 7.9×10^{28} (auf 28-29 Stellen genau)	`Decimal`
`bool`	1 Bit	`true` oder `false`	`Boolean`
`string`	unb.	Nur begrenzt durch Speicherplatz, für Unicode-Zeichenketten	`String`

Tabelle 4.1: Die Standard-Datentypen von C#

Die Standard-Datentypen bilden die Basis, es gibt aber noch weitere Datentypen, die Sie verwenden bzw. selbst deklarieren können. Doch dazu später mehr. An der Tabelle können Sie sehen, dass die hier angegebenen Datentypen eigentlich nur Aliase sind, die eigentlichen Datentypen des .NET Frameworks sind im Namensraum `System` unter den in der letzten Spalte angegebenen Namen deklariert. So ist z. B. `int` ein Alias für den Datentyp `System.Int32`.

Arten von Datentypen

In der obigen Tabelle finden sich drei Arten von Datentypen, nämlich einmal die ganzzahligen Typen (auch *Integrale Typen* genannt), dann die Gleitkommatypen und die Datentypen `string`, `bool` und `char`. `string` und `char` dienen der Aufnahme von Zeichen (`char`) bzw. Zeichenketten

(`string`), alle im Unicode-Format. Das bedeutet, jedes Zeichen belegt 2 Byte, somit können pro verwendetem Zeichensatz 65535 verschiedene Zeichen dargestellt werden. Die ersten 255 Zeichen entsprechen dabei der ASCII-Tabelle, die Sie auch im Anhang des Buchs finden. Der Datentyp `bool` entspricht einem Ja/Nein-Typ, d. h. er hat genau zwei Zustände, nämlich `true` und `false`.

Alle Standard-Datentypen der obigen Tabelle bis auf den Datentyp `string` sind Wertetypen. Da Strings nur durch die Größe des Hauptspeichers begrenzt sind, kann es sich nicht um Wertetypen handeln, denn diese haben eine festgelegte Größe. Strings hingegen sind dynamisch, d. h. hierbei handelt es sich um einen Referenztyp. Das ist auch der Grund dafür, dass ein String beliebig groß werden kann, denn die enthaltenen Daten werden nicht wie bei Wertetypen auf dem Stack angelegt, sondern auf dem Heap, der dynamisch verwaltet werden kann.

4.1.6 Type und typeof()

C# ist, wie bereits öfters angesprochen, eine typsichere Sprache. Zu den Eigenschaften einer solchen Sprache gehört auch, dass man immer ermitteln kann, welchen Datentyp eine Variable hat, oder sogar, von welcher Klasse sie abgeleitet ist. All das ist in C# problemlos möglich. Während für die Konvertierung bereits Methoden von den einzelnen Datentypen selbst implementiert werden, stellt C# für die Arbeit mit den Datentypen selbst die Klasse `Type` zur Verfügung, die im Namensraum `System` deklariert ist. Außerdem kommt der Operator `typeof` zum Einsatz, wenn es darum geht, herauszufinden, welchen Datentyp ein Objekt oder eine Variable besitzt.

typeof

Der Operator `typeof` wird eigentlich eingesetzt wie eine Methode, denn das Objekt, dessen Datentyp ermittelt werden soll, wird ihm in Klammern übergeben. Es kann sich allerdings nicht um eine Methode handeln, denn wie wir wissen, ist eine Methode immer Bestandteil einer Klasse. Ein Methodenaufruf wird immer durch die Angabe entweder des Klassennamens (bei statischen Methoden) oder des Objektnamens (bei Instanzmethoden) qualifiziert. Daran, dass dies hier nicht der Fall ist, können wir erkennen, dass es sich bei `typeof` um einen Bestandteil der Sprache selbst handeln muss.

Type

Der Rückgabewert, den `typeof` liefert, ist vom Datentyp `Type`. Dieser repräsentiert eine Typdeklaration, d. h. mit `Type` lässt sich mehr über den Datentyp eines Objekts bzw. einer Variablen herausfinden. Und auch wenn es nicht so aussieht, es gibt vieles, was man über eine Variable erfahren kann. Unter anderem liefert `Type` Methoden zur Bestimmung des

übergeordneten Datentyps, zur Bestimmung der Attribute eines Datentyps oder zum Vergleich zweier Datentypen.

Eigentlich gehen die Methoden des Datentyps `Type` *weit über Einsteigerwissen hinaus.* `Type` *ist eine »Schlüsselklasse« für die so genannte Reflection. Mittels Reflection kann mehr über Datentypen, die Dateien, in denen sie enthalten sind, Namespaces oder die Member der Datentypen (im Falle der Member handelt es sich natürlich meist um Klassen) herausgefunden werden. Reflection selbst ist ein sehr umfangreiches Thema, das in diesem Buch nicht behandelt wird; ebenso wenig werde ich detailliert auf die Möglichkeiten eingehen, die Type bietet.*

Das Visual Studio nutzt Reflection beispielsweise, um die IntelliSense-Hilfe anzuzeigen. Diese Daten werden im Hintergrund über `Type` ermittelt.

4.2 Konvertierungen in .NET

Wir haben die Typsicherheit von C# bereits angesprochen, auch die Tatsache, dass es nicht wie z. B in C++ möglich ist, einen Integer-Wert einer booleschen Variable zuzuweisen. Um dies zu verdeutlichen möchte ich nun genau dieses Beispiel – also den Vergleich zwischen C# und C++ – darstellen.

In C++ ist die folgende Zuweisung durchaus möglich:

```
/* Beispiel zur Konvertierung in C++            */
/* Diese Zuweisung funktioniert nicht in C#     */

void Test()
{
  int testVariable = 100;
  bool btest;

  btest = testVariable;
}
```

Der Wert der booleschen Variable `btest` wäre in diesem Fall `true`, weil in C++ jeder Wert größer als 0 als `true` angesehen wird. 0 ist der einzige Wert, der `false` ergibt.

In C# ist die obige Zuweisung nicht möglich. Die Datentypen `int` und `bool` unterscheiden sich in C#, daher ist eine direkte Zuweisung nicht möglich. Man müsste den Code ein wenig abändern und den booleschen Wert mittels einer Abfrage ermitteln. Dazu benutzen wir den Operator `!=` für die Abfrage auf Ungleichheit:

```
/* Beispiel zur Konvertierung in C#              */
/* Diese Zuweisung funktioniert                  */

void Test
{
  int testVariable = 100;
  bool btest;

  btest = (testVariable != 0);
}
```

In diesem Fall wird der booleschen Variable `btest` dann der Wert `true` zugewiesen, wenn die Variable `testVariable` nicht den Wert 0 hat. In C# ist diese Art der Zuweisung für einen solchen Fall unbedingt notwendig.

4.2.1 Implizite Konvertierung

Manchmal ist es jedoch notwendig, innerhalb eines Programms Werte von einem Datentyp in den anderen umzuwandeln. Hierfür gibt es in C# die *implizite* und die *explizite* Konvertierung. Außerdem stellen die Datentypen auch noch Methoden für die Konvertierung zur Verfügung. Aber der Reihe nach.

implizite Konvertierung

Wenn Sie einen Zahlenwert in einer Variable vom Typ `short` abgelegt haben, wissen Sie, dass dieser Datentyp einen gewissen Bereich beinhaltet, in dem sich der Zahlenwert befinden darf. Es ist ebenso klar, dass der Datentyp `int`, der ja einen größeren Wertebereich besitzt, ebenfalls verwendet werden könnte. Damit wird folgende Zuweisung möglich:

```
int i;
short s = 100;

i = s;
```

`i` hat den größeren Wertebereich, der in `s` gespeicherte Wert kann daher einfach aufgenommen werden. Dabei wird der Datentyp des Werts konvertiert, d. h. aus dem `short`-Wert wird automatisch ein `int`-Wert. Eine solche Konvertierung, die wir eigentlich nicht als solche wahrnehmen, bezeichnet man als *implizite Konvertierung*. Es wird dabei zwar tatsächlich eine Konvertierung vorgenommen, allerdings fällt uns das nicht auf. Den Grund dafür liefert Ihnen auf anschaulichere Weise Abbildung 4.1, die klarmacht, warum Sie nichts von der Konvertierung mitbekommen.

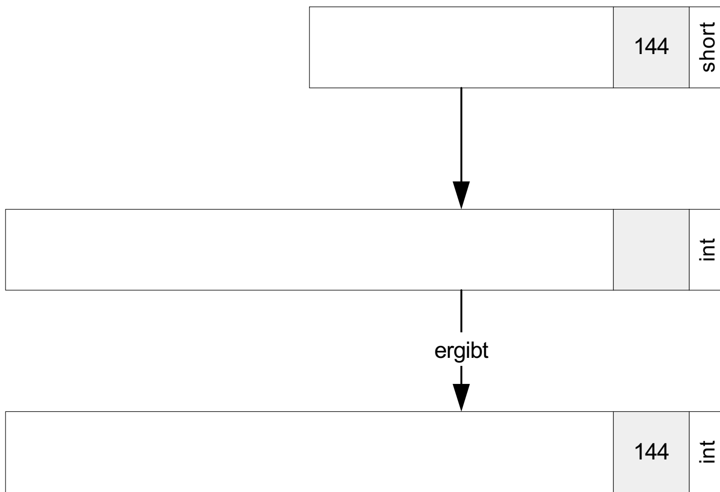

Abbildung 4.1: Implizite Konvertierung von short nach Int32

Wie aus der Abbildung zu erkennen, kann bei dieser impliziten Konvertierung kein Fehler auftreten. Der Zieldatentyp ist größer, kann also den Wert problemlos aufnehmen.

4.2.2 Explizite Konvertierung (Casting)

Ganz anders sieht es aus, wenn wir einen Wert vom Typ `int` in einen Wert vom Typ `short` konvertieren wollen. In diesem Fall ist es nicht ganz so einfach, denn der Compiler merkt natürlich, dass `int` einen größeren Wertebereich besitzt als `short`, es also zu einem Überlauf bzw. zu verfälschten Ergebnissen kommen könnte. Deswegen ist die folgende Zuweisung nicht möglich, auch wenn es von der Größe des Wertes her durchaus in Ordnung ist:

```
int i = 100;
short s;

s = i;
```

Der Compiler müsste in diesem Fall versuchen, einen großen Wertebereich in einem Datentyp mit einem kleineren Wertebereich unterzubringen. Als Vergleich: Er versucht, eine Literflasche Wasser in einem Schnapsglas unterzubringen.

Casting

Wir können nun aber dem Compiler sagen, dass der zu konvertierende Wert klein genug ist und dass er konvertieren soll. Eine solche Konvertierung wird als *explizite Konvertierung* oder auch als *Casting* bezeichnet.

Der gewünschte Zieldatentyp wird in Klammern vor den zu konvertierenden Wert oder Ausdruck geschrieben:

```
int i = 100;
short s;

s = (short)i;
```

Jetzt funktioniert auch die Konvertierung. Aus Gründen der Übersichtlichkeit wird oftmals auch der zu konvertierende Wert in Klammern gesetzt, also

```
s = (short)(i);
```

Allgemein ausgedrückt: Verwenden Sie immer die *implizite Konvertierung*, wenn der Quelldatentyp einen kleineren Wertebereich besitzt als der Zieldatentyp, und die *explizite Konvertierung*, wenn der Zieldatentyp den kleineren Wertebereich besitzt. Achten Sie aber darauf, dass Sie die eigentlichen Werte nicht zu groß werden lassen.

Eine Umwandlung ist immer dann implizit, wenn kein Fehler auftreten kann, da der Wert des Quelldatentyps immer in den Wertebereich des Zieldatentyps passt. Eine Umwandlung wird als explizit bezeichnet, wenn beim Umwandlungsvorgang ein Fehler auftreten kann, weil der Zielbereich kleiner ist als der Wertebereich des Quelldatentyps.

4.2.3 Fehler beim Casting

Sie müssen natürlich auch beim Casting darauf achten, dass der eigentliche Wert in den Wertebereich des Zieldatentyps passt. Das ist Grundvoraussetzung, denn ansonsten hilft Ihnen auch ein Casting nicht weiter. Was passieren würde, wenn der Wert zu groß wäre, sehen Sie in Abbildung 4.2.

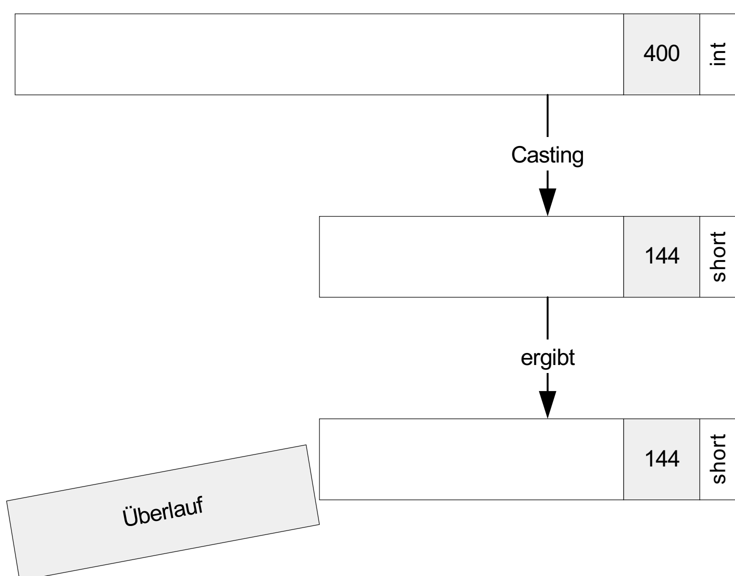

Abbildung 4.2: Fehler beim Casting mit zu großem Wert

C# würde allerdings keinen Fehler melden, lediglich der Wert wäre verfälscht. C# würde ebenso viele Bits in dem Zieldatentyp unterbringen, wie dort Platz haben, und die restlichen verwerfen. Wenn wir also den Wert 512 in einer Variablen vom Datentyp `sbyte` unterbringen wollten, der lediglich 8 Bit hat, ergäbe das den Wert 0.

Konvertierungsfehler

Um das genau zu verstehen, müssen Sie daran denken, dass der Computer lediglich mit Bits arbeitet, also mit 0 oder 1. Die unteren Bits des Werts werden problemlos in dem kleineren Datentyp untergebracht, während die oberen verloren gehen. Abbildung 4.3 veranschaulicht dies nochmals.

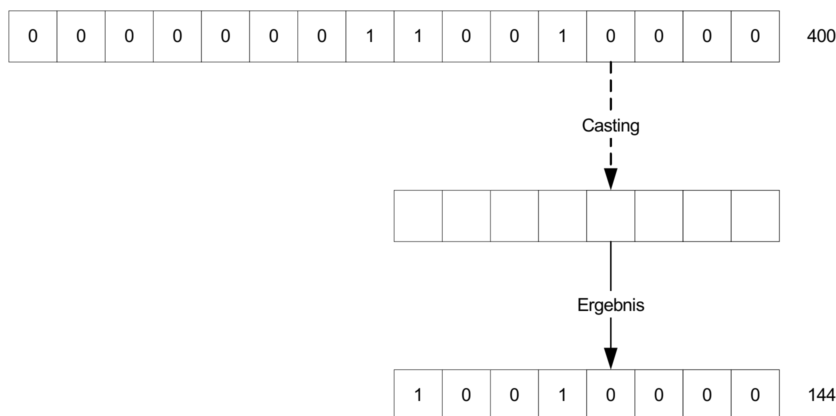

Abbildung 4.3: Fehler beim Casting mit zu großem Wert (bitweise)

4.2.4 Konvertierungsfehler erkennen

Fehler werden in C# durch so genannte *Exceptions* behandelt, die wir in *Kapitel 11* besprechen werden. Hier geht es nicht darum, wie man eine solche Exception abfängt, sondern wie man C# dazu bringt, den Fehler beim Casting zu erkennen. Wie wir gesehen haben, funktioniert das nicht automatisch, wir müssen also ein wenig nachhelfen.

checked

Um bei expliziten Konvertierungen Fehler zu entdecken (und dann auch eine Exception auszulösen), verwendet man einen speziellen Anweisungsblock, den `checked`-Block. Nehmen wir ein Beispiel, bei dem der Anwender eine Integer-Zahl eingeben kann, die dann in einen Wert vom Typ `byte` umgewandelt wird. Der Zieldatentyp hat lediglich 8 Bit zur Verfügung, der Quelldatentyp liefert 32 Bit – Damit darf die Zahl nicht größer sein als 255, sonst schlägt die Konvertierung fehl. Für diesen Fall wollen wir vorsorgen und betten die Konvertierung daher in einen `checked`-Block ein:

```
/* Programm Typumwandlung1                              */
/* Umwandlung von Datentypen mithilfe von checked      */
/* Dateiname: Typumwandlung1.cs                         */

using System;

namespace Typumwandlung1
{

  public class Beispiel
  {
    public static void Main()
    {
      int source = Convert.ToInt32(Console.ReadLine());
      byte target;
      checked
      {
        target = (byte)(source);
        Console.WriteLine("Wert: {0}",target);
      }
      Console.ReadLine();
    }
  }
}
```

Listing 4.1: Typumwandlung innerhalb eines checked-Blocks

Den Quelltext des Programms finden Sie auf der beiliegenden CD im Verzeichnis *<CDROM>:\Buchdaten\Beispiele\Kapitel_04\Typumwandlung1*.

Die Konvertierung wird nun innerhalb des `checked`-Blocks überwacht. Schlägt sie fehl, wird eine Exception ausgelöst (in diesem Fall `System.OverflowException`), die Sie wiederum abfangen können. Exceptions sind Ausnahmefehler, die ein Programm normalerweise beenden und die Sie selbst abfangen und auf die Sie reagieren können. Mehr über Exceptions erfahren Sie in *Kapitel 11*. Abbildung 4.4 verdeutlicht nochmals das Verhalten zur Laufzeit bei Verwendung eines `checked`-Blocks.

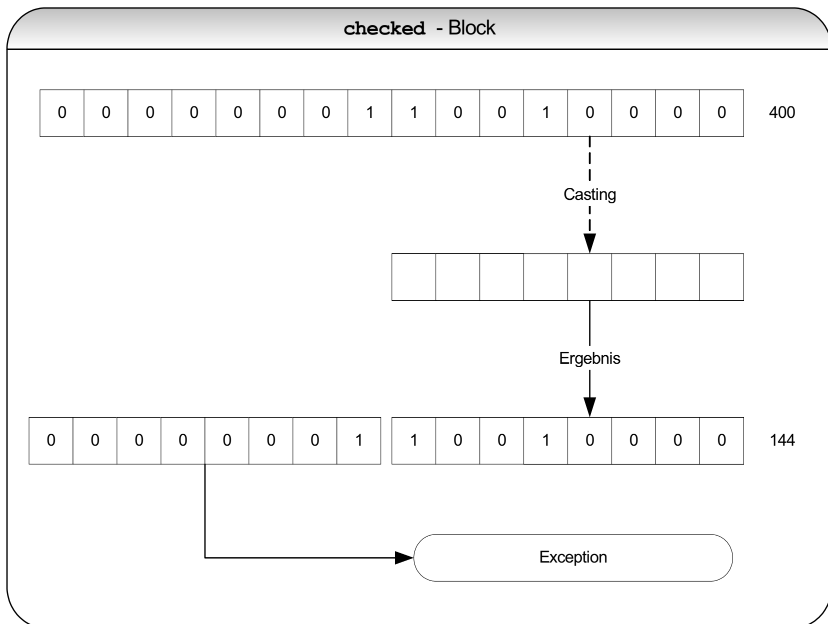

Abbildung 4.4: Exception mittels checked-Block auslösen

Die Überwachung wirkt sich aber nicht auf Methoden aus, die aus dem `checked`-Block heraus aufgerufen werden. Wenn Sie also eine Methode aufrufen, in der ebenfalls ein Casting durchgeführt wird, wird keine Exception ausgelöst. Stattdessen verhält sich das Programm wie oben beschrieben, der Wert wird verfälscht, wenn er größer ist als der maximale Bereich des Zieldatentyps. Im nächsten Beispiel wird dieses Verhalten verdeutlicht:

```
/* Programm Typumwandlung2                          */
/* Umwandlung von Datentypen mithilfe von checked   */
/* Dateiname: Typumwandlung2.cs                     */

using System;

namespace Typumwandlung2
{
```

```
  class TestClass
  {
    public byte DoCast(int theValue)
    {
      //Casting von int nach Byte
      //falscher Wert, wenn theValue>255

      return (byte)(theValue);
    }

    public void Test(int a, int b)
    {
      byte v1;
      byte v2;

      checked
      {
        v1 = (byte)(a);
        v2 = DoCast(b);
      }
    Console.WriteLine("Wert 1: {0}\nWert 2: {1}",v1,v2);
    }
  }

  class Beispiel
  {
    public static void Main()
    {
      int a,b;
      TestClass tst = new TestClass();

      Console.Write( "Wert 1 eingeben: " );
      a = Convert.ToInt32( Console.ReadLine() );
      Console.Write( "Wert 2 eingeben: " );
      b = Convert.ToInt32( Console.ReadLine() );

      tst.Test(a,b);
      Console.ReadLine();
    }
  }
}
```

Listing 4.2: Typumwandlung einmal innerhalb eines checked-Blocks und einmal über Methodenaufruf

Den Quellcode des Programms finden Sie auf der beiliegenden CD im Verzeichnis *<CDROM>:\Buchdaten\Beispiele\Kapitel_04\Typumwandlung2*.

Im Beispiel wird zweimal ein Casting durchgeführt, einmal direkt innerhalb des `checked`-Blocks und einmal in der Methode `Test()`, die aus dem `checked`-Block heraus aufgerufen wird. Wenn der erste Wert größer ist als 255, wird wie erwartet eine Exception ausgelöst. Nicht aber, wenn der zweite Wert größer ist. In diesem Fall wird die Umwandlung in der Methode `Test()` durchgeführt, eine Exception tritt nicht auf.

4.2.5 Umwandlungsmethoden

Methoden des Quelldatentyps

Wir haben nun gesehen, dass es problemlos möglich ist, Zahlenwerte in die verschiedenen numerischen Datentypen zu konvertieren. Aber was ist eigentlich, wenn wir beispielsweise eine Zahl in eine Zeichenkette konvertieren müssen, z. B. für eine Ausgabe oder weil die Methode, die wir benutzen wollen, eine Zeichenkette erwartet? Und wie sieht es umgekehrt aus, ist es auch möglich, eine Zeichenkette in eine Zahl umzuwandeln?

Ja, ist es. Aber das wissen Sie ja bereits, denn wir hatten es schon angesprochen.

In C# ist alles eine Klasse, auch die verschiedenen Datentypen sind nichts anderes als Klassen. Und als solche stellen sie natürlich Methoden zur Verfügung, die die Funktionalität beinhalten, mitunter auch Methoden für die Typumwandlung. Alle Datentypen stellen eine Methode für die Umwandlung in einen `String` zur Verfügung, der Datentyp `string` allerdings keine für die Umwandlung in einen numerischen Typ. Dafür werden entweder die statischen Methoden der Klasse `Convert` benutzt oder aber die Methode `Parse()` des Datentyps, in den konvertiert werden soll. Die Umwandlungsmethoden beginnen immer mit einem `To`, dann folgt der entsprechende Wert. Wenn Sie beispielsweise einen `string`-Wert in einen 32-Bit-`int`-Wert konvertieren möchten (vorausgesetzt, die verwendete Zeichenkette entspricht einer ganzen Zahl), verwenden Sie die Methode `Convert.ToInt32()`:

```
string myString = "125";
int    myInt;

myInt = Convert.ToInt32(myString);
```

Umgekehrt funktioniert es natürlich auch, in diesem Fall verwenden Sie die Methode `ToString()` des Datentyps:

```
string myString;
int    myInt = 125;

myString = myInt.ToString();
```

Alle Umwandlungsmethoden der Klasse `Convert` finden Sie in Tabelle 4.2.

Umwandlungsmethoden der Klasse Convert			
ToByte()	ToDecimal()	ToInt64()	ToUInt16()
ToChar()	ToDouble()	ToSByte()	ToUInt32()
ToDateTime()	ToInt16()	ToSingle()	ToUInt64()
ToBoolean()			

Tabelle 4.2: Die Umwandlungsmethoden der Klasse Convert

Bei allen Umwandlungen, ob es nun durch die entsprechenden Methoden, durch implizite Umwandlung oder durch Casting geschieht, ist immer der verwendete Datentyp zu beachten. So ist es durchaus möglich, einen Gleitkommawert in eine ganze Zahl zu konvertieren, man muss sich allerdings darüber im Klaren sein, dass dadurch die Genauigkeit verloren geht. Ebenso sieht es aus, wenn man z. B. einen Wert vom Typ `decimal` in den Datentyp `double` umwandelt, der weniger genau ist. Auch hier ist die Umwandlung zwar möglich, die Genauigkeit geht allerdings verloren.

Methoden des Zieldatentyps

Parse()

Die Umwandlung eines `String` in einen anderen Zieldatentyp, z. B. `int` oder `double`, funktioniert auch auf einem anderen Weg. Die numerischen Datentypen bieten hierfür die Methode `Parse()` an, die in mehreren überladenen Versionen existiert und grundsätzlich die Umwandlung eines `String` in den entsprechenden Datentyp veranlasst. Der Vorteil der Methode `Parse()` ist, dass zusätzlich noch angegeben werden kann, wie die Zahlen formatiert sind bzw. in welchem Format sie vorliegen. Ein einfaches Beispiel für `Parse()` liefert uns die Methode `Main()`, die es uns ermöglicht, Kommandozeilenparameter an das Programm zu übergeben:

```
/* Beispiel Typumwandlung                         */
/* Umwandlung von Kommandozeilenparametern        */

using System;

public class Beispiel
{
  public static int Main(string[] args)
  {
```

```
    // Ermitteln des ersten Zahlenwerts
    int FirstValue = 0;
    FirstValue = Int32.Parse(args[0]);

    // ... weitere Anweisungen ...
  }
```

Das Array `args[]` enthält die an das Programm in der Kommandozeile übergebenen Parameter. Was es mit Arrays auf sich hat, werden wir in *Kapitel 7* noch ein wenig näher betrachten, für den Moment soll genügen, dass es sich dabei um ein Feld mit Daten handelt, die alle den gleichen Typ haben und über einen Index angesprochen werden können.

Kommandozeilen-parameter

Andere Programmiersprachen besitzen ebenfalls die Möglichkeit, Parameter aus der Kommandozeile an das Programm zu übergeben. Es gibt jedoch einen Unterschied: In C# wird der Name des Programms nicht mit übergeben, d. h. das erste Argument, das Sie in `Main()` auswerten können, ist auch wirklich der erste übergebene Kommandozeilenparameter.

Im obigen Fall erwartet das Programm eine ganze Zahl vom Typ `int`. Die Methode `Parse()` wird benutzt, um den `String` in einen Integer umzuwandeln. Dabei hat diese Methode wie ebenfalls schon angesprochen den Vorteil, nicht nur den Zahlenwert einfach so zu konvertieren, sondern auch landesspezifische Einstellungen zu berücksichtigen. Für die herkömmliche Konvertierung ist die Methode `ToInt32()` der Klasse `Convert` absolut ausreichend:

```
/* Beispiel Typumwandlung                          */
/* Umwandlung von Kommandozeilenparametern         */

using System;

public class Beispiel
{
  public static int Main(string[] args)
  {
    // Ermitteln des ersten Zahlenwerts
    int FirstValue = 0;
    FirstValue = Convert.ToInt32(args[0]);

    // ... weitere Anweisungen ...
  }
```

Der Datentyp `string` ist ein recht universeller Datentyp, der sehr häufig in Programmen verwendet wird. Aus diesem Grund werden wir uns diesem Datentyp in einem gesonderten Abschnitt zuwenden.

4.3 Boxing und Unboxing

Wir haben bereits gelernt, dass es zwei Arten von Daten gibt, Referenztypen und Wertetypen. Möglicherweise haben Sie sich bereits gefragt, warum man mit Wertetypen ebenso umgehen kann wie mit Referenztypen, wo es sich doch um zwei unterschiedliche Arten des Zugriffs handelt bzw. die Daten auf unterschiedliche Art im Speicher des Computers abgelegt sind. Der Trick bzw. das Feature, das C# hier verwendet, heißt *Boxing*.

Boxing

Wenn ein Wertetyp als Referenztyp verwendet werden soll, werden die enthaltenen Daten sozusagen verpackt. C# benutzt dafür den Datentyp `object`, der bekanntlich die Basis aller Datentypen darstellt und somit auch jeden Datentyp aufnehmen kann. Im Unterschied zu anderen Sprachen merkt sich `object` aber, welcher Art von Daten in ihm gespeichert sind, um eine Konvertierung in die andere Richtung ebenfalls zu ermöglichen.

Mit diesem Objekt, bei dem es sich nun um einen Referenztyp handelt, ist das Weiterarbeiten problemlos möglich. Umgekehrt können Sie einen auf diese Art und Weise umgewandelten Wert auch wieder in einen Wertetyp zurückkonvertieren.

Falls Sie sich fragen, wozu dieses Boxing und das dazugehörige Unboxing denn notwendig sein sollte … nun, die Basisklasse aller Klassen und Datentypen im .NET Framework ist `object`*. Damit sind Methoden, deren Parameter vom Typ* `object` *sind, universell einsetzbar. Gäbe es kein Boxing, könnte man solchen Methoden keinen Wertetyp übergeben, weil ein Referenztyp gefordert wird. Da es Boxing aber gibt, nimmt die Methode jeden Datentyp an und der Compiler »boxed« den übergebenen Wert automatisch.*

4.3.1 Boxing

Sie können Boxing und das Gegenstück Unboxing auch selbst in Ihren Applikationen anwenden. Der folgende Code speichert den Wert einer `int`-Variable in einer Variable vom Typ `object`, also einem Referenztyp.

```
/* Programm Boxing1                                    */
/* Boxing eines Wertetyps                              */
/* Dateiname: Boxing1.cs                               */

using System;

namespace Boxing1
{
  public class TestClass
  {
    public static void Main()
    {
      int i = 100;
      object o;
      o = i;  //Boxing !!
      Console.WriteLine("Wert ist {0}.",o);
      Console.ReadLine();
    }
  }
}
```

Listing 4.3: Boxing eines Wertetyps

Der Wert von `i` ist nun in einem Objekt `o` gespeichert. Grafisch dargestellt sieht das dann so aus, wie in Abbildung 4.5. Die Ausgabe des Programms entspricht der Ausgabe des Wertes von `i`:

```
Wert ist 100.
```

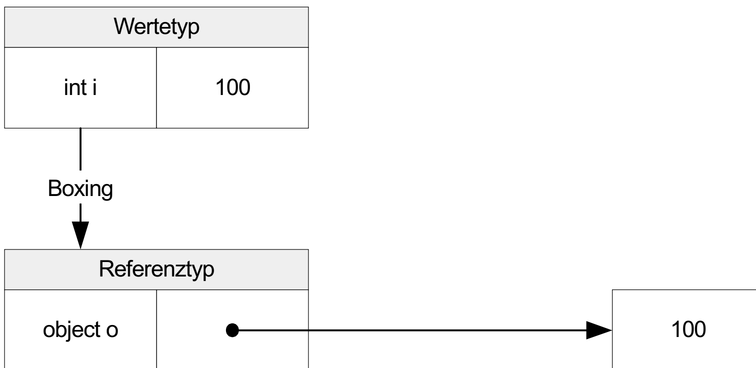

Abbildung 4.5: Boxing eines Integer-Werts

Den Quelltext des Programms finden Sie auf der beiliegenden CD im Verzeichnis *<CDROM>:\Buchdaten\Beispiele\Kapitel_04\Boxing1*.

4.3.2 Unboxing

Der umgekehrte Weg ist zwar vom Prinzip her ebenso einfach, allerdings muss der Datentyp, in den das Objekt zurückkonvertiert werden soll, bekannt sein. Es ist nicht möglich, ein Objekt, das einen `int`-Wert enthält, in einen `byte`-Wert umzuwandeln. Hier zeigt sich wieder die Typsicherheit von C#.

```
/* Programm Boxing2                                   */
/* Unboxing eines Wertetyps mit Fehler                */
/* Dateiname: Boxing2.cs                              */

using System;

namespace Boxing2
{
  public class TestClass
  {
    public static void Main()
    {
      int i = 100;
      object o;
      o = i;  //Boxing !!
      Console.WriteLine("Wert ist {0}.",o);

      //Rückkonvertierung
      byte b = (byte)(o); //funktioniert nicht!!
      Console.WriteLine("Byte-Wert: {0}",b);
    }
  }
}
```

Listing 4.4: Unboxing eines Wertetyps mit Fehler

Den Quellcode finden Sie auf der beiliegenden CD im Verzeichnis *<CDROM>:\Buchdaten\Beispiele\Kapitel_04\Boxing2*.

Obwohl die Größe des in `o` enthaltenen Werts durchaus in eine Variable vom Typ `byte` passen würde, ist dieses Unboxing nicht möglich. Im Objekt `o` ist der enthaltene Datentyp mit gespeichert, damit verlangt C# beim Unboxing, dass auch hier ein `int`-Wert für die Rückkonvertierung verwendet wird.

Im Falle des Beispiels aus Abbildung 4.4 wird eine `InvalidCastException` ausgelöst, die besagt, dass eine solche Umwandlung nicht möglich ist.

Wir haben jedoch bereits die andere Möglichkeit der Typumwandlung, die explizite Umwandlung oder das Casting, kennen gelernt. Wenn eine implizite Konvertierung nicht funktioniert, sollte es doch eigentlich mit einer expliziten Konvertierung funktionieren. Das tut es auch, das folgende Beispiel beweist es.

```
/* Programm Boxing3                              */
/* Unboxing eines Wertetyps mit Casting          */
/* Dateiname: Boxing3.cs                         */

using System;

namespace Boxing3
{
  public class TestClass
  {
    public static void Main()
    {
      int i = 100;
      object o;
      o = i;  //Boxing !!
      Console.WriteLine("Wert ist {0}.",o);

      //Rückkonvertierung
      byte b = (byte)((int)(o)); //funktioniert!!
      Console.WriteLine("Byte-Wert: {0}.",b);
      Console.ReadLine();
    }
  }
}
```

Listing 4.5: Unboxing mit Casting

Den Quellcode des Programms finden Sie auf der beiliegenden CD im Verzeichnis *<CDROM>:\Buchdaten\Beispiele\Kapitel_04\Boxing3*.

In diesem Beispiel wird der in `o` enthaltene Wert zunächst in einen `int`-Wert zurückkonvertiert, wonach aber unmittelbar das Casting zu einem `byte`-Wert folgt. Und da der enthaltene Wert nicht zu groß für den Datentyp `byte` ist, ergibt sich als Ausgabe:

```
Wert ist 100.
Byte-Wert: 100.
```

Beim Boxing wird ein Wertetyp in einen Referenztyp »verpackt«. Anders als in diversen anderen Programmiersprachen merkt sich das Objekt in C# aber, welcher Datentyp darin verpackt wurde. Damit ist ein Unboxing nur in den gleichen Datentyp möglich.

4.3.3 Den Datentyp ermitteln

Der im Objekt `o` enthaltene Datentyp kann auch ermittelt werden. `o` stellt dafür die Methode `GetType()` zur Verfügung, die den Typ der enthaltenen Daten zurückliefert. Der Ergebnistyp von `GetType()` ist ein Objekt vom Typ `Type`. Und da `Type` wie jedes andere Objekt auch eine Methode `ToString()` enthält, ist es mit folgender Konstruktion möglich, den Datentyp als `string` auszugeben:

```
/* Programm Boxing4                                  */
/* Ermittlung des Datentyps eines geboxeten Wertes  */
/* Dateiname: Boxing4.cs                             */

using System;

namespace Boxing4
{
  public class TestClass
  {
    public static void Main()
    {
      int i = 100;
      object o;
      o = i;  //Boxing !!
      Console.WriteLine("Wert ist {0}.",o);
      Console.WriteLine("Datentyp ist: {0}",
        o.GetType().ToString());

      Console.ReadLine();
    }
  }
}
```

Listing 4.6: Ermittlung des Datentyps eines geboxeten Wertes

Den Quelltext des Programms finden Sie auf der beiliegenden CD im Verzeichnis *<CDROM>:\Buchdaten\Beispiele\Kapitel_04\Boxing2.*

Damit hätten wir Boxing soweit abgehandelt. Normalerweise werden Sie es in Ihren Applikationen dem .NET Framework überlassen, das Boxing durchzuführen. Es funktioniert ja auch automatisch und problemlos. Manuelles Boxing oder Unboxing ist in den seltensten Fällen nötig, aber wie Sie sehen auch nicht besonders schwierig.

4.4 Strings

Der Datentyp `string` ist ein recht universell einsetzbarer Datentyp, den wir auch schon in einem Beispiel benutzt haben. Strings sind Zeichenketten, d. h. eine Variable vom Typ `string` kann jedes beliebige Zeichen aufnehmen. Weiterhin bietet auch dieser Datentyp mehrere Funktionen zum Arbeiten mit Zeichenketten.

`string` weist auch noch eine andere Besonderheit auf. Obwohl die Deklaration wie bei einem Wertetyp funktioniert, handelt es sich doch um einen Referenztyp, denn eine Variable vom Typ `string` kann so viele Zeichen aufnehmen, wie Platz im Speicher ist. Damit ist die Größe einer `string`-Variablen nicht festgelegt, der verwendete Speicher muss dynamisch (auf dem Heap) reserviert werden. Der Datentyp `string` ist (zusammen mit `object`) der einzige Basisdatentyp, der ein Referenztyp ist. Alle anderen Basistypen sind Wertetypen.

4.4.1 Unicode und ASCII

ASCII

Der ASCII-Zeichensatz (*American Standard Code for Information Interchange*) war der erste Zeichensatz auf einem Computer. Anfangs arbeitete man noch mit einem 7-Bit-ASCII-Zeichensatz, wodurch 127 Zeichen darstellbar waren. Das genügte für alle Zeichen des amerikanischen Alphabets. Später jedoch wurde der Zeichensatz auf 8 Bit Breite ausgebaut, um die Sonderzeichen der meisten europäischen Sprachen ebenfalls aufnehmen zu können, und für die meisten Anwendungen genügte dies auch. Unter Windows konnte man sich den Zeichensatz aussuchen, der für das entsprechende Land passend war, und ihn benutzen.

In Zeiten, da das Internet eine immer größere Rolle spielt, und zwar sowohl bei der Informationsbeschaffung als auch bei der Programmierung, genügt ein Byte nicht mehr, um alle Zeichen darzustellen. Genauer gesagt: Wenn jemand auf eine Internet-Seite zugreifen will, muss dafür auch der Zeichensatz installiert sein, mit dem diese Seite arbeitet. Uns als Europäern fällt das nicht besonders auf, meist bewegen wir uns auf deutschen oder englischen Seiten, bei denen der Zeichensatz ohnehin zum größten Teil übereinstimmt. Was aber, wenn wir auf eine japanische oder chinesische Seite zugreifen wollen? In diesem Fall sehen wir auf dem Bildschirm nicht die entsprechenden Schriftzeichen, sondern in den meisten Fällen einen Mischmasch aus Sonderzeichen ohne irgendetwas lesen zu können. Um es noch genauer zu sagen: Auch ein Chinese hätte durchaus Probleme, seine Sprache wieder zu erkennen.

Unicode

C# wurde von Microsoft als eine Sprache angekündigt, die die Anwendungsentwicklung sowohl für das Web als auch für lokale Computer vereinfachen soll. Gerade bei der Entwicklung von Internetapplikationen ist es aber sehr wichtig, dass es keine Konflikte mit dem Zeichensatz gibt. Deshalb arbeitet C# komplett mit dem *Unicode*-Zeichensatz, bei dem ein Zeichen nicht durch ein Byte, sondern durch zwei Bytes repräsentiert wird.

Der Unterschied ist größer, als man denkt. Waren mit 8 Bit noch 2^7 Zeichen (= 255 Zeichen) darstellbar, sind es jetzt 2^{15} Zeichen (= 65535 Zeichen). Diese Anzahl genügt, um alle Zeichen aller Sprachen dieser Welt und noch einige Sonderzeichen unterzubringen. Um die Größenordnung noch deutlicher darzustellen: Etwa ein Drittel des Unicode-Zeichensatzes ist noch unbelegt.

C# arbeitet komplett mit dem Unicode-Zeichensatz. Sowohl was die Strings innerhalb Ihres eigenen Programms angeht als auch was die Quelltexte betrifft, auch hier wird der Unicode-Zeichensatz verwendet. Theoretisch ist also jedes Zeichen darstellbar. Allerdings gilt für die Programmierung nach wie vor nur der englische (bzw. amerikanische) Zeichensatz mit den bekannten Sonderzeichen. Eine Variable mit dem Bezeichner `Zähler` ist leider nicht möglich. Der Grund hierfür ist allerdings auch offensichtlich: Immerhin soll mit der Programmiersprache in jedem Land gearbeitet werden können, somit muss man einen kleinsten Nenner finden. Und bezüglich des Zeichensatzes ist das nun mal der amerikanische Zeichensatz.

4.4.2 Standard-Zuweisungen

Zeichenketten werden immer in doppelten Anführungszeichen angegeben. Die folgenden Zuweisung an eine Variable vom Datentyp `string` wäre also der Normalfall:

```
string myString = "Hallo Welt";
```

oder natürlich

```
string myString;
myString = "Hallo Welt";
```

Es gibt aber noch eine weitere Möglichkeit, einem `String` einen Wert zuzuweisen. Wenn Sie den Inhalt eines bereits existierenden `String` kopieren möchten, können Sie die statische Methode `Copy()` verwenden und den Inhalt eines bestehenden `String` an den neuen `String` zuweisen:

```
string myString = "Frank Eller";
string myStr2   = string.Copy(myString);
```

Ebenso ist es möglich, nur einen Teilstring zuzuweisen. Dazu wird eine Instanzmethode des erzeugten Stringobjekts verwendet:

```
string myString = "Frank Eller";
string myStr2   = myString.Substring(6);
```

Die Methode `Substring()` kopiert einen Teil des bereits bestehenden `String myString` in den neu erstellten `myStr2`. `Substring()` ist eine überladene Methode, Sie können entweder den Anfangs- und Endpunkt der Kopieraktion angeben oder nur den Anfangspunkt, also den Index des Zeichens, bei dem die Kopieraktion begonnen werden soll. Denken Sie daran, dass immer bei 0 mit der Zählung begonnen wird, d. h. das siebte Zeichen hat den Index 6. Wenn Sie die zweite Variante benutzen, wird der gesamte `String` bis zum Ende kopiert.

Zusätzlich zu diesen Möglichkeiten gibt es noch erweiterte Zuweisungsmöglichkeiten. Ebenso wie bei der Ausgabe durch `WriteLine()` gelten z. B. auch bei Strings die Escape-Sequenzen, denn `WriteLine()` tut ja nichts anderes, als den String, den Sie angeben, zu interpretieren und auszugeben.

4.4.3 Erweiterte Zuweisungsmöglichkeiten

Escape-Sequenzen

Kommen wir hier zunächst zu den bereits angesprochenen Escape-Sequenzen. Diese können natürlich auch hier vollständig benutzt werden. So können Sie z. B. auf folgende Art einen String dazu bringen, doppelte Anführungszeichen auszugeben:

```
string myString = "Dieser Text hat \"Anführungszeichen\".";
Console.WriteLine(myString);
```

Die Ausgabe wäre dann entsprechend:

```
Dieser Text hat "Anführungszeichen".
```

Literalzeichen

Alle anderen Escape-Sequenzen, die Sie bereits kennen gelernt haben, sind ebenfalls möglich. Allerdings benötigen Sie diese nicht, um Sonderzeichen darstellen zu können. In C# haben Sie Strings betreffend noch eine weitere Möglichkeit, nämlich die, die Escape-Sequenzen nicht zu bearbeiten. Ein Beispiel soll deutlich machen, wozu dies gut sein kann.

Nehmen wir an, Sie wollten einen Pfad zu einer bestimmten Datei in einem String speichern. Das kommt durchaus öfter vor, z. B. wenn Sie in Ihrem Programm die letzte verwendete Datei speichern wollen. Sobald Sie jedoch den Backslash als Zeichen benutzen, wird das von C# als Escape-Sequenz betrachtet, woraus folgt, dass Sie für jeden Backslash im Pfad eben zwei Backslashes hintereinander schreiben müssen:

```
string myString = "d:\\aw\\csharp\\Kapitel5\\Kap05.doc";
```

Das @-Zeichen

Einfacher wäre es, wenn in diesem Fall die Escape-Sequenzen nicht bearbeitet würden, wir also den Backslash nur einmal schreiben müssten. Das würde im Übrigen auch der normalen Schreibweise entsprechen. Immerhin können wir nicht verlangen, wenn ein Anwender einen Dateinamen eingibt, dass dieser jeden Backslash doppelt schreibt. Um die Bearbeitung der Escape-Sequenzen zu verhindern schreiben wir vor den eigentlichen String einfach ein @-Zeichen:

```
string myString = @"d:\aw\csharp\Kapitel5\Kap05.doc";
```

Fortan werden die Escape-Sequenzen nicht mehr bearbeitet, es genügt jetzt, einen Backslash zu schreiben.

Sonderzeichen

Sie werden sich möglicherweise fragen, wie Sie in einem solchen `String` ohne Escape-Sequenz z. B. ein doppeltes Anführungszeichen schreiben. Denn die oben angesprochene Möglichkeit existiert ja nicht mehr, der Backslash würde als solcher angesehen und das darauf folgende doppelte Anführungszeichen würde das Ende des `String` bedeuten. Die Lösung ist ganz einfach: Schreiben Sie solche Sonderzeichen einfach doppelt:

```
string myString = "Das sind ""Anführungszeichen"".";
```

mehrzeilige Strings

Das @-Zeichen birgt noch eine weitere Besonderheit. Sie wissen, dass das Ende einer C#-Anweisung durch das Semikolon angegeben wird und dass Sie dadurch in der Lage sind, Anweisungen auf mehrere Zeilen zu verteilen. Das funktioniert aber nicht, wenn Sie z.B. einen String zuweisen. Hier ist es notwendig, den String in jeder Zeile zu beenden und mit einem weiteren String in der nächsten Zeile zu verbinden.

```
String aString = "Das ist ein String, der über "+
"mehrere Zeilen geht und daher mithilfe von " +
"+-Zeichen zusammengesetzt werden muss";
```

Wollen Sie nun einen wirklich mehrzeiligen String erzeugen, benötigen Sie einen Zeilenumbruch. Den erreichen Sie über "\r\n":

```
String aString = "Das ist ein String, der über \r\n"+
"mehrere Zeilen geht und daher mithilfe von \r\n" +
"+-Zeichen zusammengesetzt werden muss";
```

Sehen Sie sich nun den folgenden Befehl an:

```
String aString = @"Das ist ein String, der über
mehrere Zeilen geht. Dabei wird das @-Zeichen benutzt
um anzuzeigen, dass die Zeilenumbrüche Teil des
Strings sein sollen.";
```

Ich werde an dieser Stelle nicht fragen, ob Sie glauben, dass das funktioniert. Geben Sie das Ganze ein und lassen Sie den String ausgeben. Oder schauen Sie sich Abbildung 4.6 an.

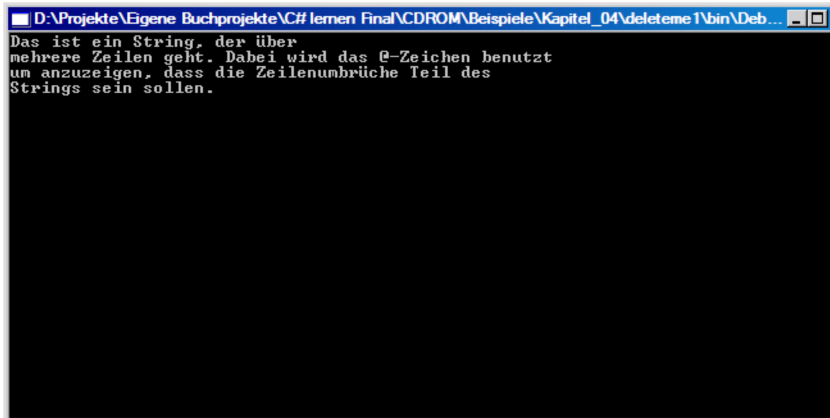

Abbildung 4.6: Die Ausgabe des Strings

Wie Sie sehen, ermöglicht es das @-Zeichen, dass die Zeilenumbrüche mit in den String aufgenommen werden. Auf diese Weise können Sie also auch mehrzeilige Strings erzeugen.

4.4.4 Zugriff auf Strings

Um auf einen `String` zuzugreifen gibt es mehrere Möglichkeiten. Die eine Möglichkeit besteht darin, den gesamten `String` zu benutzen, wie wir es oftmals tun. Eine weitere Möglichkeit, die wir auch schon kennen gelernt haben, ist die, auf einen Teilstring zuzugreifen (mittels der Methode `Substring()`). Es existiert aber noch eine Variante.

Strings sind Zeichenketten. Wenn man diesen Begriff wörtlich nimmt, sind Strings tatsächlich aneinander gereihte Zeichen. Der Datentyp für ein Zeichen ist `char`. Damit kann auf einen `String` auch zeichenweise zugegriffen werden.

Die Eigenschaft `Length` eines `String` liefert dessen Länge zurück. Wir könnten also eine `for`-Schleife benutzen, um alle Zeichen eines `String` zu kontrollieren. Die `for`-Schleife haben wir zwar noch nicht behandelt, in diesem Fall werde ich aber dem entsprechenden Kapitel ein wenig vorgreifen und die `for`-Schleife hier schon benutzen. Auf die genaue Funktionsweise werden wir in *Kapitel 5* noch eingehen. Ebenso vorgreifen werde ich auf die `if`-Anweisung, die eine Verzweigung bewirkt. Auch die werden wir in *Kapitel 5* genauer betrachten.

Mit Hilfe der for-Schleife können wir einen Programmblock mehrfach durchlaufen. Zum Zählen wird eine Variable benutzt, die wir dazu verwenden können, jedes Zeichen des String einzeln auszuwerten.

```
/* Programm Strings1                              */
/* Zugriff auf die Zeichen eines Strings          */
/* Dateiname: Strings1.cs                         */

using System;

namespace Strings1
{
  class TestClass
  {
    public static void Main()
    {
      string myStr = "Hallo Welt.";
      string xStr  = "";

      for (int i=0;i<myStr.Length;i++)
      {
        char x = myStr[i];
        if (!(x=='e'))
          xStr += x;
      }

      Console.WriteLine(xStr);
      Console.ReadLine();
    }
  }
}
```

Listing 4.7: Zugriff auf Strings mithilfe einer for-Schleife

Wenn Sie dieses Programm ausführen, ergibt sich als Ausgabe

```
Hallo Wlt.
```

Sie finden den Quellcode des Programms auf der beiliegenden CD, im Verzeichnis *<CDROM>:\Buchdaten\Beispiele\Kapitel_04\Strings1*.

Wir kontrollieren jedes Zeichen des String darauf, ob es sich um ein »e« handelt. Ist dies der Fall, tun wir nichts, ansonsten fügen wir den Buchstaben unserem zweiten String hinzu. Da die Abfrage myStr[i] den Datentyp char zurückliefert, müssen wir auch mit einem Wert des Typs char vergleichen. Würden wir an dieser Stelle das e in doppelte Anführungszeichen schreiben, hätten wir allerdings den Datentyp string.

Sie sehen am Quellcode, dass das »e« für den Vergleich in einfache Anführungsstriche eingeschlossen ist. Damit signalisieren wir dem Compiler, dass er diesen Buchstaben nicht als `string`-Datentyp behandeln soll, sondern als einzelnes Zeichen vom Typ `char`.

Wenn Sie ein Zeichen als `char` *verwenden wollen, schließen Sie es einfach in einfache Anführungszeichen ein. Wenn Sie es in doppelte Anführungszeichen einschließen, wird es vom Compiler wie ein String behandelt. Im Beispiel aus Abbildung 4.7 wäre es dann nötig gewesen, die Variable* `myChar` *in den Datentyp* `string` *zu konvertieren. So geht es aber einfacher.*

Rechenoperatoren

Sicherlich haben Sie außerdem im obigen Beispiel bemerkt, dass hier ein Rechenoperator auf einen String angewendet wurde, nämlich der Operator +=. Tatsächlich ist es so, dass der +-Operator sowie der +=-Operator auch auf Strings angewendet werden können und zwei Strings zu einem zusammenfügen:

```
string myString1 = "Frank";
string myString2 = " Eller";
string myString  = myString1+myString2;
```

In diesem Fall würde `myString` den Wert »Frank Eller« enthalten. Das Gleiche erledigt diese Anweisung:

```
string myString = "Frank";
myString += " Eller";
```

4.4.5 Methoden von string

Der Datentyp `string` ist wie jeder andere Datentyp in C# auch eine Klasse und stellt somit Methoden zur Verfügung, die Sie im Zusammenhang mit Zeichenketten verwenden können. In den beiden folgenden Listen werden einige häufig verwendete Methoden des Datentyps `string` vorgestellt.

4.4.6 Klassenmethoden von string

Die Klassenmethoden von `string` sind allesamt überladene Methoden mit relativ vielen Aufrufmöglichkeiten. An dieser Stelle werde ich mich daher auf die wichtigsten bzw. am häufigsten verwendeten Methoden beschränken.

```
public static bool Compare(
    string strA
    string strB
);
```

```
public static bool Compare(
    string strA
    string strB
    bool ignoreCase
);
```

Mit diesen Versionen der Methode `Compare()` können zwei komplette Zeichenketten miteinander verglichen werden. Anhand des Parameter `ignoreCase` können Sie angeben, ob die Groß-/Kleinschreibung ignoriert werden soll oder nicht.

```
public static bool Compare(
    string strA;
    int indexA;
    string strB;
    int indexB;
    int length
);
public static bool Compare(
    string strA;
    int indexA;
    string strB;
    int indexB;
    int length;
    bool ignoreCase
);
```

Mit diesen Versionen der Methode `Compare()` können Sie vergleichen, ob bestimmte Teile zweier Zeichenketten übereinstimmen. Auch hier ist es wieder möglich, die Groß-/Kleinschreibung zu ignorieren.

```
public static string Concat(object);
public static string Concat(string[] values);
public static string Concat(object[] args);
```

Die Methode `Concat()` dient dem Zusammenfügen mehrerer Zeichenketten bzw. Objekte, die Zeichenketten repräsentieren.

```
public static string Copy(string str0);
```

Die Methode `Copy()` liefert eine Kopie des übergebenen `String` zurück.

```
public static bool Equals(
    string a,
    string b
);
```

Die Methode `Equals()` kontrolliert, ob die beiden übergebenen Strings gleich sind. Auf Groß-/Kleinschreibung wird bei diesem Vergleich geachtet. Sind beide Strings gleich, wird `true` zurückgeliefert, sind sie es

nicht oder ist einer der übergebenen Strings ohne Zuweisung (null), wird false zurückgeliefert.

```
public static string Format(
    string format,
    object arg0
);
public static string Format(
    string format,
    object[] args
);
```

Die Methode Format() ermöglicht die Formatierung von Werten. Dabei enthält der übergebene Parameter format den zu formatierenden String mit Platzhaltern und Formatierungszeichen, der Parameter arg0 enthält das Objekt, das an der Stelle des Platzhalters eingefügt und entsprechend der Angaben formatiert wird. Mehr über die Format()-Funktion erfahren Sie in *Kapitel 4.5*.

Instanzmethoden von string

```
public object Clone();
```

Die Methode Clone() liefert die aktuelle String-Instanz als Objekt zurück.

```
public int CompareTo(object o);
public int CompareTo(string s);
```

Die Methode CompareTo() vergleicht die aktuelle String-Instanz mit dem als Parameter übergebenen Objekt bzw. String. Zurückgeliefert wird ein Integer-Wert, der angibt, wie die beiden Strings sich zueinander verhalten. Grundlage für den Vergleich ist das Alphabet, wobei ein String als umso kleiner angesehen wird, je weiter er im Vergleich Richtung Anfang angeordnet würde. Der Rückgabewert ist kleiner 0, wenn die aktuelle String-Instanz kleiner als der Parameter ist, gleich 0, wenn sie gleich dem Parameter ist, und größer 0 oder 1, wenn sie größer als der Parameter ist.

```
public bool EndsWith(string value);
```

Die Methode EndsWith() kontrolliert, ob der übergebene Parameter dem Ende der aktuellen String-Instanz entspricht. Wenn value länger ist als die aktuelle Instanz oder nicht dem letzten Teil entspricht, wird false zurückgeliefert.

```
public new bool Equals(string value);
public override bool Equals(object obj);
```

Die Methode `Equals()` existiert auch als Instanzmethode, funktioniert aber genauso wie die entsprechende statische Methode. Allerdings ist in diesem Fall der erste `String`, mit dem verglichen werden soll, bereits durch die aktuelle Instanz vorgegeben.

```
public Type GetType();
```

Die Methode `GetType()` ist in allen Klassen enthalten, also auch in der Klasse `string`. Sie liefert den Datentyp zurück.

```
public int IndexOf(char[] value);
public int IndexOf(string value);
public int IndexOf(char value);
public int IndexOf(
    string value,
    int startIndex
);
public int IndexOf(
    char[] value,
    int startIndex
);
public int IndexOf(
    char value,
    < startIndex
);
public int IndexOf(
    string value,
    int startIndex,
    int endIndex
);
public int IndexOf(
    char[] value,
    int startIndex,
    int endIndex
);
public int IndexOf(
    char value,
    int startIndex,
    int endIndex
);
```

Die Methode `IndexOf()` ermittelt den Offset, an dem der angegebene Teilstring in der aktuellen `String`-Instanz auftritt. Wenn der Teilstring überhaupt nicht enthalten ist, wird der Wert –1 zurückgeliefert. Mit den Parametern `startIndex` bzw. `endIndex` kann auch noch ein Bereich festgelegt werden, in dem die Suche stattfinden soll.

Analog zur Methode `IndexOf()` existiert eine Methode `LastIndexOf()`, die das letzte Auftreten des angegebenen Teilstring zurückliefert. Sie existiert in den gleichen überladenen Versionen wie die Methode `IndexOf()`.

```
public string Insert(
    int startIndex,
    string value
);
```

Die Methode `Insert()` fügt einen Teilstring an der angegebenen Stelle in den aktuellen `String` ein und liefert das Ergebnis als `string` zurück.

```
public string PadLeft(int totalWidth);
public string PadLeft(
    int totalWidth,
    char paddingChar
);
```

Die Methode `PadLeft()` richtet einen `String` rechtsbündig aus und füllt ihn von vorne mit Leerstellen, bis die durch den Parameter `totalLength` angegebene Gesamtlänge erreicht ist. Falls gewünscht, kann über den Parameter `paddingChar` auch ein Zeichen angegeben werden, mit dem aufgefüllt wird.

```
public string PadRight(int totalWidth);
public string PadRight(
    int totalWidth,
    char paddingChar
);
```

Die Methode `PadRight()` verhält sich wie die Methode `PadLeft()`, nur dass der `String` jetzt linksbündig ausgerichtet wird.

```
public string Remove(
    int startIndex,
    int Count
);
```

Die Methode `Remove()` entfernt einen Teil aus der aktuellen `String`-Instanz. Die Anfangsposition und die Anzahl der Zeichen, die entfernt werden sollen, können Sie mit den Parametern `startIndex` und `count` angeben.

```
public string Replace(
    char oldChar,
    char newChar
);
```

Die Methode `Replace()` ersetzt im gesamten aktuellen `String` ein Zeichen gegen ein anderes.

```
public string[] Split(char[] separator);
public string[] Split(
    char[] separator,
    int count
);
```

Die Methode `Split()` teilt einen `String` in diverse Teilstrings auf. Als Trennzeichen wird das im Parameter `separator` angegebene Array aus Zeichen benutzt. Mit dem Parameter `count` können Sie zusätzlich die maximale Zahl an Teilstrings, die zurückgeliefert werden sollen, angeben.

```
public bool StartsWith(string value);
```

Die Methode `StartsWith()` kontrolliert, ob die aktuelle `String`-Instanz mit dem im Parameter value angegebenen Teilstring beginnt.

```
public string SubString(int startIndex)
public string SubString(
    int startIndex
    int length
);
```

Die Methode `Substring()` liefert einen Teilstring der aktuellen `String`-Instanz zurück, wobei Sie die Anfangsposition und die Länge angeben können. Wird die Länge nicht angegeben, geht der Teilstring bis zum Ende.

```
public string Trim()
public string Trim(char[] trimChars)
```

Die Methode `Trim()` entfernt standardmäßig alle Leerzeichen am Anfang und am Ende der aktuellen `String`-Instanz und liefert den resultierenden Teilstring zurück. Wollen Sie statt der Leerzeichen andere Zeichen entfernen, können Sie diese im Parameter `trimChars` angeben.

```
public string TrimEnd(char[] trimChars);
```

Die Methode `TrimEnd()` entfernt alle angegebenen Zeichen am Ende des `String`. Wenn Sie für den Parameter `trimChars` den Wert `null` übergeben, werden alle Leerzeichen entfernt.

```
public string TrimStart(char[] trimChars)
```

Die Methode `TrimStart()` entfernt alle im Array `trimChars` angegebenen Zeichen am Anfang der aktuellen `String`-Instanz und liefert die resultierende Zeichenkette zurück.

4.5 Formatierung von Daten

4.5.1 Standardformate

Sie haben bereits im *Hallo-Welt*-Programm mit Platzhaltern gearbeitet. Was ausgegeben wurde, war nichts anderes als eine feste Zeichenkette, also im Prinzip auch nur ein String. Sie haben allerdings über diese Platzhalter auch die Möglichkeit, die Ausgabe numerischer Werte zu formatieren.

Format

Die Art und Weise, wie das passiert, ist ein wenig komplexer. Zum Einsatz dieser Möglichkeiten soll uns an dieser Stelle genügen, dass die Methode `ToString()`, die jeder Datentyp implementiert, bei den Wertetypen auch zur Formatierung der Ausgabe geeignet ist. Alternativ können Sie auch die `Format()`-Methode der Klasse `System.String` verwenden. Auch `Console.WriteLine()` verwendet die Formatierungsmöglichkeit.

Formatzeichen

Die Angabe, welche Art von Formatierung gewünscht ist, geschieht im Platzhalter durch die Angabe eines Formatzeichens und ggf. einer Präzisionsangabe für die Anzahl der Stellen, die ausgegeben werden sollen. Ein Beispiel soll verdeutlichen, wie das funktioniert. Wir wollen zwei eingegebene Integer-Werte so formatieren, dass sie korrekt untereinander stehen. Dazu geben wir im Platzhalter an, dass alle numerischen Werte mit fünf Stellen ausgegeben werden sollen. Nimmt eine Zahl die Stellen nicht komplett ein, wird mit Nullen aufgefüllt.

```
/* Programm Formatierung1*/
/* Formatierung von Zahlenwerten bei der Ausgabe     */
/* Dateiname: Formatierung1.cs                       */

using System;

namespace Formatierung1
{
  public class Beispiel
  {
    public static void Main()
    {
      int a,b;
      Console.Write("Geben Sie Zahl 1 ein: ");
      a = Convert.ToInt32(Console.ReadLine());
      Console.Write("Geben Sie Zahl 2 ein: ");
      b = Convert.ToInt32(Console.ReadLine());

      Console.WriteLine("Die Zahlen lauten:");
      Console.WriteLine("Zahl 1: {0:D5}",a);
```

```
      Console.WriteLine("Zahl 2: {0:D5}",b);
      Console.ReadLine();
    }
  }
}
```

Listing 4.8: Formatierung von Zahlen bei der Ausgabe auf die Konsole

Bei einer Angabe zweier Zahlen 75 und 1024 würde die Ausgabe folgendermaßen aussehen:

```
Die Zahlen lauten
Zahl 1: 00075
Zahl 2: 01024
```

Die Zahlen stehen damit exakt untereinander.

Den Quelltext des Programms finden Sie auf der beiliegenden CD im Verzeichnis *<CDROM>:\Buchdaten\Beispiele\Kapitel_04\Formatierung1.*

Die Angabe der Präzision hat jedoch unterschiedliche Bedeutung. Im Falle einer Gleitkommazahl würde nämlich nicht die Anzahl der Gesamtstellen angegeben, sondern die Anzahl der Nachkommastellen. Falls notwendig, rundet C# hier auch automatisch auf oder ab, um die geforderte Genauigkeit zu erreichen. Im Falle einer Hexadezimalausgabe würde eine ganze Zahl auch automatisch in das Hexadezimal-Format umgewandelt. Das Formatierungszeichen für das Hexadezimalformat ist *X*:

```
/* Programm Formatierung2                          */
/* Formatierung von Zahlenwerten bei der Ausgabe   */
/* Dateiname: Formatierung2.cs                     */

using System;

namespace Formatierung2
{
  public class Beispiel
  {
    public static void Main()
    {
      int a,b;
      Console.Write("Geben Sie Zahl 1 ein: ");
      a = Convert.ToInt32(Console.ReadLine());
      Console.Write("Geben Sie Zahl 2 ein: ");
      b = Convert.ToInt32(Console.ReadLine());

      Console.WriteLine("Die Zahlen lauten:");
      Console.WriteLine("Zahl 1: {0:X4}",a);
```

```
      Console.WriteLine("Zahl 2: {0:X4}",b);
      Console.ReadLine();
    }
  }
}
```

Listing 4.9: Formatierung zweier Zahlenwerte als Hexadezimalzahl

Bei einer Eingabe der Zahlen 75 und 1024 würde sich in diesem Beispiel folgende Ausgabe ergeben:

```
Die Zahlen lauten
Zahl 1: 004B
Zahl 2: 0400
```

Den Quelltext des Programms finden Sie auf der beiliegenden CD im Verzeichnis *<CDROM>:\Buchdaten\Beispiele\Kapitel_04\Formatierung2*.

Tabelle 4.3 gibt Ihnen eine Übersicht über die verschiedenen Formatierungszeichen und ihre Bedeutung.

Zeichen	Formatierung
C,c	Währung (engl. *Currency*), formatiert den angegebenen Wert als Preis unter Verwendung der landesspezifischen Einstellungen.
D,d	Dezimalzahl (engl. *Decimal*), formatiert einen Gleitkommawert. Die Präzisionszahl gibt die Anzahl der Nachkommastellen an.
E,e	Exponential (engl. *Exponential*), wissenschaftliche Notation. Die Präzisionszahl gibt die Nummer der Dezimalstellen an. Bei wissenschaftlicher Notation wird immer mit einer Stelle vor dem Komma gearbeitet. Der Buchstabe »E« im ausgegebenen Wert steht für »mal 10 hoch«.
F,f	Gleitkommazahl (engl. *fixed Point*), formatiert den angegebenen Wert als Zahl mit der durch die Präzisionsangabe festgelegten Anzahl an Nachkommastellen.
G,g	Kompaktformat (engl. *General*), formatiert den angegebenen Wert entweder als Gleitkommzahl oder in wissenschaftlicher Notation. Ausschlaggebend ist, welches der Formate die kompaktere Darstellung ermöglicht.
N,n	Numerisch (engl. *Number*), formatiert die angegebene Zahl als Gleitkommazahl mit Kommas als Tausender-Trennzeichen. Das Dezimalzeichen ist der Punkt.
X,x	Hexadezimal, formatiert den angegebenen Wert als hexadezimale Zahl. Der Präzisionswert gibt die Anzahl der Stellen an. Eine angegebene Zahl im Dezimalformat wird automatisch ins Hexadezimalformat umgewandelt.

Tabelle 4.3: Die Formatierungszeichen von C#

4.5.2 Selbst definierte Formate

Sie haben nicht nur die Möglichkeit, die Standardformate zu benutzen. Sie können die Ausgabe auch etwas direkter steuern, indem Sie in einem selbst definierten Format die Anzahl der Stellen und die Art der Ausgabe festlegen. Tabelle 4.4 listet die verwendeten Zeichen auf.

Zeichen	Verwendung
#	Platzhalter für eine führende oder nachfolgende Leerstelle
0	Platzhalter für eine führende oder nachfolgende 0
.	Der Punkt gibt die Position des Dezimalpunkts an.
,	Jedes Komma gibt die Position eines Tausendertrenners an.
%	Ermöglicht die Ausgabe als Prozentzahl, wobei die angegebene Zahl mit 100 multipliziert wird
E+0 E-0	Das Auftreten von E+0 oder E-0 nach einer 0 oder nach dem Platzhalter für eine Leerstelle bewirkt die Ausgabe des Werts in wissenschaftlicher Notation.
;	Das Semikolon wirkt als Trenner für Zahlen, die entweder größer, gleich oder kleiner 0 sind. Die erste Formatierungsangabe bezieht sich auf positive Werte, die zweite auf den Wert 0 und die dritte auf negative Werte. Werden nur zwei Sektionen angegeben, gilt die erste Formatierungsangabe sowohl für positive Zahlen als auch für den Wert 0.
\	Der Backslash bewirkt, dass das nachfolgende Zeichen so ausgegeben wird, wie Sie es in den Formatierungsstring schreiben. Es wirkt nicht als Formatierungszeichen.
'	Wollen Sie mehrere Zeichen ausgeben, die nicht als Teil der Formatierung angesehen werden, können Sie diese in einfache Anführungszeichen setzen.

Tabelle 4.4: Formatierungszeichen für selbst definierte Formate

Ein Beispiel soll auch hier verdeutlichen, wie Sie mit diesen Zeichen arbeiten. Wir wollen eine Zahl im Währungsformat ausgeben, wobei wir die deutsche Währung dahinter schreiben. Ist die Zahl negativ, soll sie in Klammern gesetzt werden. Außerdem verwenden wir Tausendertrennzeichen.

```
/* Programm Formatierung3                               */
/* Formatierung von Zahlenwerten bei der Ausgabe        */
/* Dateiname: Formatierung3.cs                          */

using System;

namespace Formatierung3
{
  public class Beispiel
  {
```

```
    public static void Main()
    {
      double a = 0;
      Console.Write("Geben Sie eine Zahl ein: ");
      a = Convert.ToDouble(Console.ReadLine());

      Console.WriteLine(
        "Formatiert: {0:#,#.00' EUR';(#,#.00)' EUR'}",
      a);

      Console.ReadLine();
    }
  }
}
```

Listing 4.10: Formatierung eines Währungswertes

Sie finden den Quelltext des Programms auf der beiliegenden CD im Verzeichnis *<CDROM>:\Buchdaten\Beispiele\Kapitel_04\Formatierung3*.

Bitte beachten Sie, dass Sie als Trennzeichen sowohl das Komma als auch den Punkt verwenden können. Da wir hier mit Währungen arbeiten, bedeutet das Komma die Trennung zwischen Euro und Cent, der Punkt ein Tausendertrennzeichen. Die Eingabe von

```
22.50
```

ergibt damit folgende Ausgabe:

```
2.250,00 EUR
```

4.6 Zusammenfassung

In diesem Kapitel haben wir uns mit den verschiedenen Standard-Datentypen beschäftigt, die C# zur Verfügung stellt. Es ging dabei nicht nur darum, welche Datentypen es gibt, sondern auch darum, wie man damit arbeitet und z. B. Werte von einem in den anderen Datentyp konvertiert. Besonders behandelt haben wir in diesem Zusammenhang den Datentyp `string`, der eine Sonderstellung einnimmt.

Strings sind ein recht universeller Datentyp und werden daher auch sehr häufig benutzt. Deshalb ist es auch sinnvoll, mehr über diesen Datentyp zu erfahren. Mit Hilfe von Strings ist es möglich, Zeichenketten zu verwalten und auch andere Daten zu formatieren.

In diesem Zusammenhang haben wir uns auch nochmals den Platzhaltern zugewendet und verschiedene Möglichkeiten der Formatierung unterschiedlicher Datentypen durchgesprochen.

4.7 Kontrollfragen

Auch für dieses Kapitel habe ich wieder einen Satz Fragen zusammengestellt, der das bisher erworbene Wissen ein wenig vertiefen soll. Gehen Sie die Fragen sorgfältig durch, die Antworten finden Sie im letzten Kapitel.

1. Welcher Standard-Datentyp ist für die Verwaltung von 32-Bit-Ganzzahlen zuständig?
2. Was ist der Unterschied zwischen impliziter und expliziter Konvertierung?
3. Wozu dient ein `checked`-Programmblock?
4. Wie wird die explizite Konvertierung auch genannt?
5. Worin besteht der Unterschied zwischen den Methoden `Parse()` und `Convert.ToInt32()` bezogen auf die Konvertierung eines Werts vom Typ `string`?
6. Wie viele Bytes belegt ein Buchstabe innerhalb eines Strings?
7. Was wird verändert, wenn das Zeichen @ bei einem String verwendet wird?
8. Welche Escape-Sequenz dient dazu, einen Wagenrücklauf durchzuführen (und gleichzeitig eine Zeile weiter zu schalten)?
9. Was bewirkt die Methode `Concat()` des Datentyps `string`?
10. Was bewirkt das Zeichen # bei der Formatierung eines Strings?
11. Wie können mehrere Zeichen innerhalb einer Formatierungssequenz exakt so ausgegeben werden, wie sie geschrieben sind?
12. Was bewirkt die Angabe des Buchstabens `G` im Platzhalter bei der Formatierung einer Zahl, wie z. B. in `{0:G5}`?

4.8 Übungen

In diesen Übungen beschäftigen wir uns mit Zahlen und deren Darstellung. Natürlich werden wir das im vorigen Kapitel Gelernte nicht außer Acht lassen.

Übung 1

Erstellen Sie eine neue Klasse mit zwei Feldern, die `int`-Werte aufnehmen können. Stellen Sie Methoden zur Verfügung, mit denen diese Werte ausgegeben und eingelesen werden können. Standardmäßig soll der Wert der Felder 0 sein.

Übung 2

Schreiben Sie eine Methode, in der Sie die beiden Werte dividieren. Das Ergebnis soll aber als `double`-Wert zurückgeliefert werden.

Übung 3

Schreiben Sie eine Methode, die das Gleiche tut, den Wert aber mit drei Nachkommastellen und als String zurückliefert. Die vorherige Methode soll weiterhin existieren und verfügbar sein.

Übung 4

Schreiben Sie eine Methode, die zwei `double`-Werte als Parameter übernimmt, beide miteinander multipliziert, das Ergebnis aber als `int`-Wert zurückliefert. Die Nachkommastellen dürfen einfach abgeschnitten werden.

Übung 5

Schreiben Sie eine Methode, die zwei als `int` übergebene Parameter dividiert. Das Ergebnis soll als `short`-Wert zurückgeliefert werden. Falls die Konvertierung nach `short` nicht funktioniert, soll das abgefangen werden. Überladen Sie die bestehenden Methoden zum Dividieren der Werte in den Feldern der Klasse.

Übung 6

Schreiben Sie eine Methode, die zwei `string`-Werte zusammenfügt und das Ergebnis als `string`, rechtsbündig, mit insgesamt 20 Zeichen, zurückliefert. Erstellen Sie für diese Methode eine eigene Klasse und sorgen Sie dafür, dass die Methode immer verfügbar ist.